2019 IEEE/ACM Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)

HPC container runtimes have minimal or no performance impact

Alfred Torrez, Timothy Randles, Reid Priedhorsky
High Performance Computing Division
Los Alamos National Laboratory
Los Alamos, NM, USA
{atorrez,reidpr,trandles}@lanl.gov

***Abstract*—HPC centers are facing increasing demand for greater software flexibility to support faster and more diverse innovation in computational scientific work. Containers, which use Linux kernel features to allow a user to substitute their own software stack for that installed on the host, are an increasingly popular method to provide this flexibility. Because standard container technologies such as Docker are unsuitable for HPC, three HPC-specific technologies have emerged: Charliecloud, Shifter, and Singularity.**

A common concern is that containers may introduce performance overhead. To our knowledge, no comprehensive, rigorous, HPC-focused assessment of container performance has previously been performed. Our present experiment compares the performance of all three HPC container implementations and bare metal on multiple dimensions using industry-standard benchmarks (SysBench, STREAM, and HPCG).

We found no meaningful performance differences between the four environments, with the possible exception of modest variation in memory usage.

These results suggest that HPC users should feel free to containerize their applications without concern about performance degradation, regardless of the container technology used. It is an encouraging development towards greater adoption of user-defined software stacks to increase the flexibility of HPC systems.

I. Introduction

HPC centers have developed broad and deep expertise at providing the large-scale compute needed to support a specific type of science: MPI-based simulation of physical systems. Recently, additional fields such as data science and non-MPI simulation are becoming increasingly compute-focused. HPC centers are able to provide the scale of compute needed but have lacked the flexibility to serve well these non-traditional users, who use novel-to-HPC parallel frameworks, applications, and Linux distributions.

One way to deliver this flexibility is containers, which use Linux system calls to allow user-defined software stacks (UDSS) up to and including a complete Linux distribution [1]. There are many container implementations. The industry standard is Docker, but it is unsuitable for HPC for various scaling and security reasons [1], [2]. Accordingly, three HPC-focused container implementations have emerged: Charliecloud [1],[1] Shifter [2], and Singularity [3].

The standard concern for any HPC-related innovation is how it affects performance, and containers face the same question. Specifically, any abstraction layer creates opportunities to impose performance overhead, and an alternate UDSS technology, virtual machines, has clear impact [4], [5].

Existing work suggests a rough consensus that containers have a low to minimal performance overhead for HPC applications. However, it is clearly possible to have non-trivial overhead [5], and there are anecdotal reports of containers performing *better* than bare metal.

Our goal in this work is to evaluate the performance overhead of all three HPC-focused container implementations in a rigorous, comprehensive, HPC-scale manner, in order to provide more dependable answers on HPC container performance overhead. We evaluated performance on four dimensions:

- **CPU performance** using SysBench,[2] a suite of microbenchmarks that measure various aspects of performance.
- **Memory performance** using STREAM,[3] which measures sustained memory bandwidth for simple memory-bound vector kernels.
- **Application performance** using the High Performance Conjugate Gradients (HPCG) Benchmark,[4] a relatively new benchmark developed as a complement to the High Performance Linpack (HPL)[5] benchmark but enhanced to more accurately reflect current HPC applications.
- **Memory usage** by sampling the STREAM and HPCG tests as they ran.

The three performance tests yielded no significant differences, while memory usage showed non-zero but minimal container overhead. Our key conclusions are:

1) Performance overhead may not just be low, but actually zero. The differences that exist seem to be related to differing environments (e.g., linking differences or lack of userspace hardware libraries), rather than the container runtime or kernel code. (Note that *all* Linux processes are in namespaces, which are the key kernel construct

This work was supported in part by the Advanced Simulation and Computing Program; the Exascale Computing Project under project number 17-SC-20-S; and the LANL Institutional Computing Program, which is supported by the U.S. Department of Energy National Nuclear Security Administration under contract 89233218CNA000001. LA-UR 19-28946

[1]Disclosure: The authors are members of the Charliecloud team.

[2]https://github.com/akopytov/sysbench/
[3]https://www.cs.virginia.edu/stream/
[4]http://www.hpcg-benchmark.org/
[5]https://www.netlib.org/benchmark/hpl/

978-1-7281-6028-3/19/$31.00 ©2019 IEEE
DOI 10.1109/CANOPIE-HPC49598.2019.00010

for containers; they might just be the root namespaces at the top of the tree.)

2) Container implementations seem not to matter for performance. That is, performance is not a reason to choose one implementation over another.

These results place HPC containers on a firmer foundation. Users should feel free to containerize if it serves their science, and system administrators should feel free to offer containerization as a solution to user demand for flexibility.

The remainder of this paper details related work, our methods, and the results We close briefly by proposing implications of our results and future work.

II. Related work

The question of container performance for HPC has been addressed a modest number of times in the literature. We focus here on evaluations either of HPC applications, on HPC systems, or using HPC container implementations. An additional literature on generic container performance also exists; see for example [6], [7], [8].

Xavier et al. evaluated LINPACK, a CFD benchmark, and three generic benchmarks on Linux-VServer, OpenVZ, and LXC up to 4 nodes (64 cores), finding performance close to native on most of the tests [4]. (At the time in 2013, no HPC-specific container implementations existed.)

Our previous work introducing Charliecloud contains basic MPI benchmarks up to 128 nodes (2048 cores); we found "minimal overhead" [1]. Brayford et al. tested TensorFlow on an HPC cluster under Charliecloud up to 32 nodes (1536 cores), finding "no performance overhead for AI workloads" and "negligible memory overhead" [9].

Jacobsen and Canon used Shifter to find that encapsulating a container image in a single filesystem image file yielded fairly dramatic gains in Python application load time, measured using the Pynamic benchmark on an unspecified number of nodes [2]. This is a valuable result, but a different performance focus than the present work.

Singularity has attracted more attention. Under this implementation, Kovács tested SysBench and IPerf (a network benchmark) using a single node (16 cores), finding CPU "almost at the level of the native performance" and network that "closely approximate[s] the performance of the native execution" [10]. Le and Paz tested MPI benchmarks on an unspecified number of nodes and a neuron simulation up to 8 nodes (192 cores), finding only "a small margin" for MPI but "more overhead" for the application [11]. Younge et al. tested HPGC and an MPI benchmark up to 32 nodes (768 cores), finding that Singularity "does not impact network performance" and "provide[s] native performance" for HPCG [5]. Wang, Evans, and Huang tested four applications up to 18 nodes (864 cores), finding "almost no effect on the performance" [12].

Considering this body of work together, we identify three themes: (a) one HPC container implementation at a time, (b) diversity of benchmarks, and (c) lack of HPC-level scale. In the present work, we attempt to close this gap by evaluating all three HPC containers implementations in one rigorous series of experiments up to 512 nodes (18,432 cores). Our findings of no or minimal performance impact are consistent with the rough consensus above, and we offer some additional insights regarding this consensus below.

III. Methods

This section details our experimental procedures. Note that this work was done on production HPC systems, not under laboratory conditions, in order to better reflect probable real-world results.

A. Hardware

We ran our tests on LANL's CTS-1 clusters Grizzly (1490 nodes, 128 GiB RAM/node) and Fog (32 nodes, 256 GiB RAM/node). These nodes have 36 CPU cores on an Intel S2600KP motherboard with 2× Intel E5-2695v4 (Broadwell) 2.1 GHz 18-core CPUs; hyperthreading is disabled. Node DIMMs are 16 GiB 2400MHz DDR4. The cluster interconnect is Intel OmniPath OP HFI single-port connected via PCIe-Gen3 ×16. Grizzly is a 2:1 oversubscribed fat-tree topology, while Fog is flat.

B. Operating system and container images

Our site uses NNSA's Tri-Lab Operating System Stack (TOSS) version 3.4-4, which is based on RHEL 7.6 and Linux kernel 3.10.0-957.5.1. This operating system is typical of HPC centers, which tend to be RHEL-based. We also point out that the 3.10 kernel version is somewhat misleading, as this kernel includes a large number of Red Hat patches and backports and is quite a bit different from upstream 3.10. Key software components are Charliecloud 0.9.10, Shifter 18.03.0, Singularity 3.3.0-rc1, HPCG 3.0, STREAM 5.10, SysBench 1.0.17, OpenMPI 3.1.4, and GCC 4.8.5.

The host (and thus the bare metal environment) used the full 68 GiB TOSS stack, which is both excessively large for containerization and mostly irrelevant to our experiments. Thus, for the container images, we removed several hundred packages, yielding 1.2 GiB container images. These images were almost, but not quite, effectively identical to the bare metal stack; we note some meaningful differences below. We built one container image with Docker and converted it to each technology using its native tools.

Singularity allows configuration of namespaces used for containers. We used the default configuration.

C. CPU performance

We used SysBench to time a computation of the prime numbers below 40 million, using all 36 hardware threads. We ran 100 single-node tests for each of the four environments (bare metal and the three container technologies). We had exclusive access to a Fog node and rotated round-robin between the environments.

D. Memory performance

We ran 100 single-threaded STREAM tests per environment per kernel — *copy*, *scale*, *add*, and *triad* — compiled with STREAM_ARRAY_SIZE set to 2 billion to match the recommended 4× cache. STREAM process were pinned to core 23 using Slurm argument --cpu_bind=v,core,map_cpu:23. (A pilot experiment suggested that core 23 provided the most consistent memory access time.) We had exclusive access to a Grizzly node and rotated round-robin between the environments.

E. Application performance

We ran two separate HPCG test groups. For both, we used a cube dimension of 104 and a run time of 60 seconds, all 36 node cores, one MPI rank per core, and one thread per rank. We set the HPCG run time to 60 seconds; HPCG reported that "the result is valid", which is consistent with our site benchmarking expertise. Each technology rotated the same number of times through the same sets of nodes, in order to reduce the impact of variable node performance. We had exclusive access to the entire machine and ran multiple tests simultaneously; that is, our tests shared resources with one another but not other users.

In the first group, we ran HPCG with power-of-two node counts from 1 to 512 on Grizzly. The number of HPCG runs per condition varied because our reservation contained a bad node, invalidating several runs. Node counts from 1 to 32 all had 4 runs per condition, while node counts 64 and up varied from 4 to 10 runs, with the exception of Charliecloud at 64 nodes, which had 3.

In the second, we ran HPCG on power-of-two node counts from 1 to 32 on Fog, with 50 runs per condition. (Recall that Fog and Grizzly have the same specifications, except that Fog has many fewer nodes and more memory per node.)

F. Memory usage

We wanted three measurements related to memory:

1) Total note memory used, which is the actual constraint.
2) Incremental node memory used by the container implementation, e.g. ancillary processes, which reflects overhead in a more focused way.
3) Memory used by the actual application, which tells us whether the application's memory behavior has been changed by containerization.

We addressed the first two using STREAM, assuming that the simpler benchmark would allow better insight into container implementation behavior. We measured total node memory consumption during the experiments described in Section III-D above, using MemTotal minus MemFree from /proc/meminfo, sampled at 10-second intervals. Incremental node memory usage was addressed with the same samples; we removed background usage by subtracting the first sample, which was taken at the beginning of the job before starting STREAM, from the others. This measured memory used by the container implementation plus STREAM itself, without the operating system, other supporting processes, or the unpacked image.

We addressed the third using the Group 1 HPCG runs, assuming that a more complex benchmark would be more likely to see memory behavior altered by containerization. Again we sampled at 10-second intervals, but using the pmap command, which reports memory usage at the process level. We used the writeable/private field, which counts only process-private memory, factoring out memory shared with other processes.

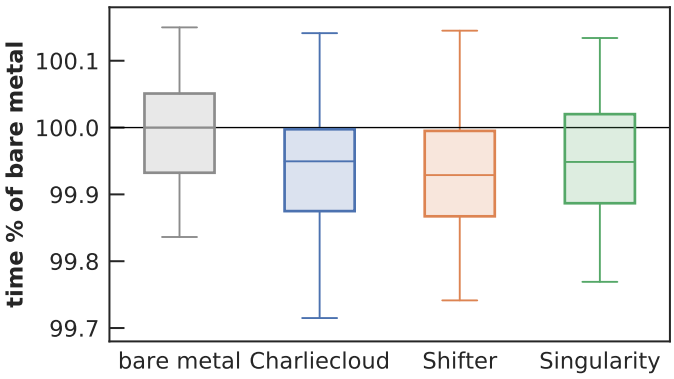

Fig. 1. SysBench prime number computation time relative to median bare metal performance of 129.36 seconds; lower is better. Boxes show the median and middle 50%, while whiskers show the maximum and minimum. The four environments showed essentially identical performance.

IV. Results

A. CPU performance

Figure 1 shows SysBench prime number computation time; these numbers were reported by the benchmark itself, not external timing. Performance was nearly identical across all four environments. Median performance had a spread of 0.07% (129.27 to 129.36 seconds), while all 400 runs were contained within a spread of 0.4% (128.99 to 129.55 seconds).

As an example of the subtle issues that can plague container performance tests on production systems, our first try at this experiment yielded containers that performed almost identically, but bare metal was about 1.8% slower. It turned out that the bare metal build of SysBench found and linked against libaio, which was not present in our trimmed-down TOSS container image. We rebuilt SysBench with --disable-aio and re-ran the experiment, and the difference disappeared.

B. Memory performance

Figure 2 shows histograms of STREAM memory performance. Again, performance was nearly identical across all four environments; the histograms overlap almost completely.

C. Application performance

Figure 3 shows Group 1 HPCG performance (as reported by the benchmark itself) on node counts from 1 to 512. Percentages are normalized to median bare metal performance at each node count, e.g. a value of 101 indicates 1% faster than median bare metal and 99 indicates 1% slower.

While the four environments have similar performance, the spread is higher than the prior two experiments, up to roughly 3%. We also made a single run of each technology at 1024 nodes (36,864 nodes), omitted from the figure due to the small sample size. The results were: bare metal 6.73 TFLOPS,

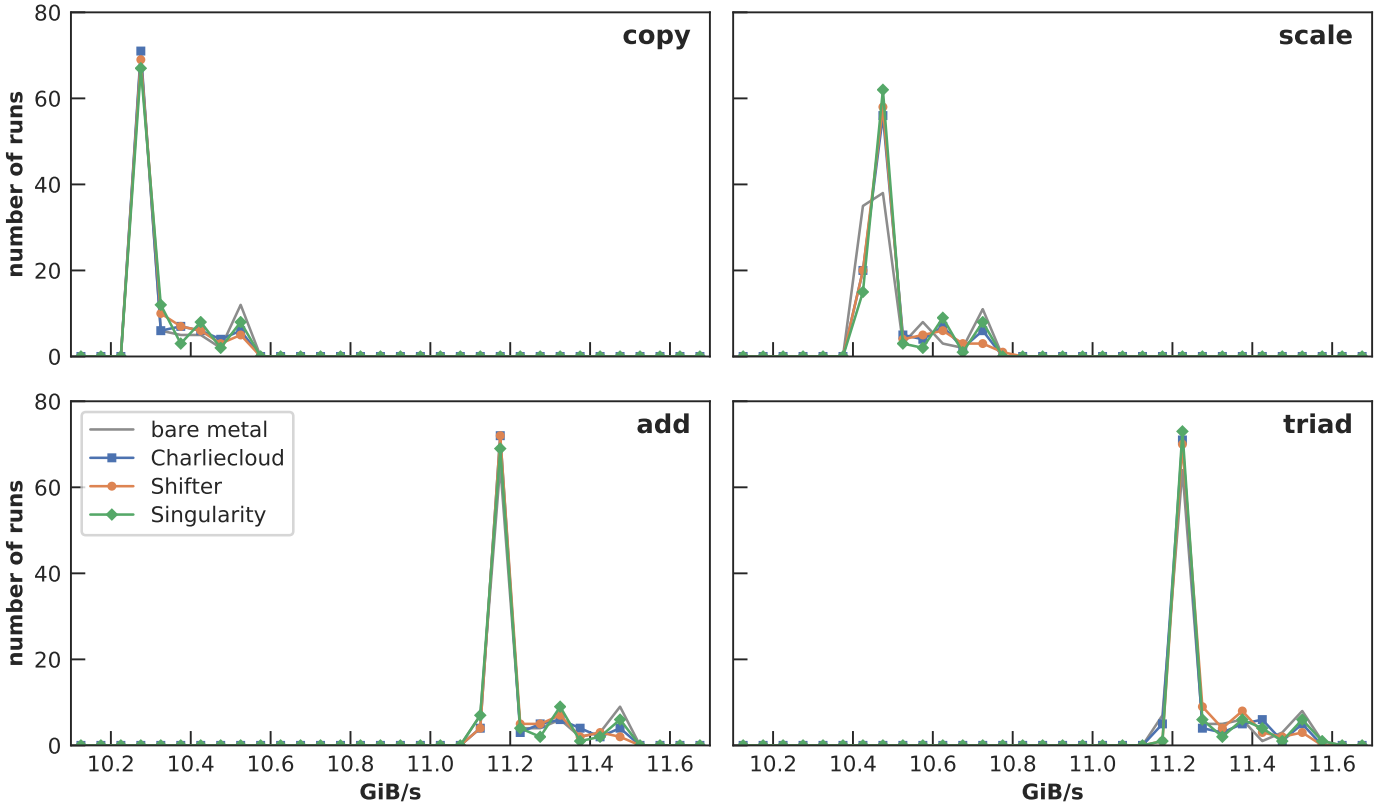

Fig. 2. STREAM memory bandwidth histogram; rightward is better. The four environments showed essentially identical performance.

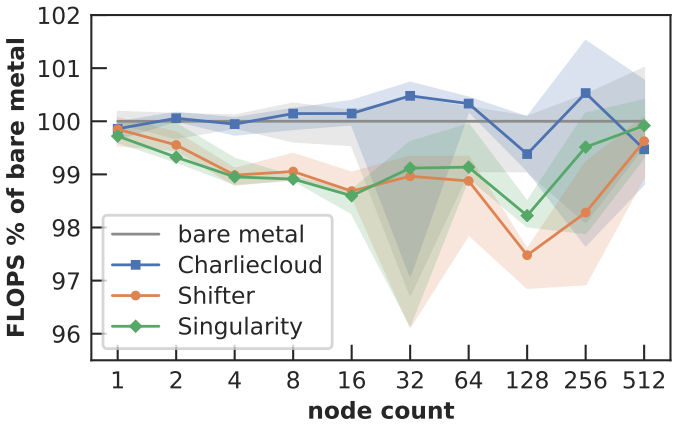

Fig. 3. Group 1 HPCG benchmark performance as percent of bare metal median; higher is better. The lines show median performance, and the shaded areas minimum to maximum. The four environments are similar in performance, but not nearly as close as the above results. While the Shifter and Singularity are lower than the bare metal and Charliecloud lines for much of the graph, we believe this gap is spurious, as discussed in the text.

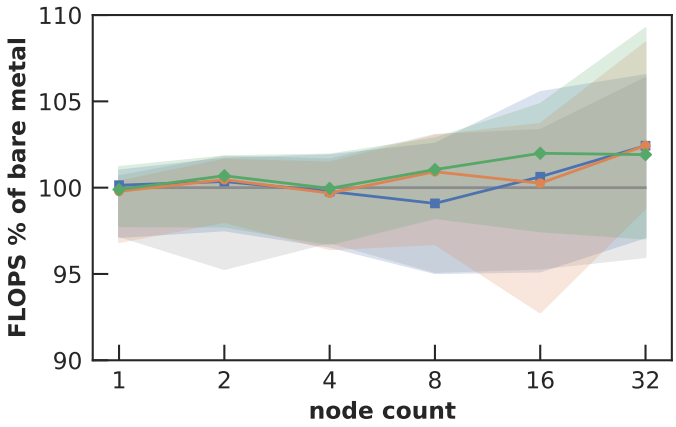

Fig. 4. Group 2 HPCG performance. The three container technologies track one another very closely, while bare metal is a little slower at 32 nodes. We again believe this difference is spurious, as discussed in the text.

Charliecloud 6.91, Shifter 6.84, Singularity 7.03, a spread of 4.5%. These results are consistent with a hypothesis of no meaningful difference between the environments.

However, we were skeptical that the two obvious clusters (bare metal / Charliecloud and Shifter / Singularity) were real, for two reasons: (a) relatively small number of samples and (b) an inadvertent inconsistency in the experiment setup. We used PMIx for all 256- and 512-node runs, and all Shifter and Singularity runs, but PMI2 for Charliecloud and bare metal from 1 to 128 nodes.

The purpose of Group 2 was to resolve this question. We made 50 runs of each condition using PMI2 throughout. Figure 4 shows resulting performance. The three container technologies track one another very closely. Bare metal is a little slower (about 2%) at 32 nodes. However, we did discover

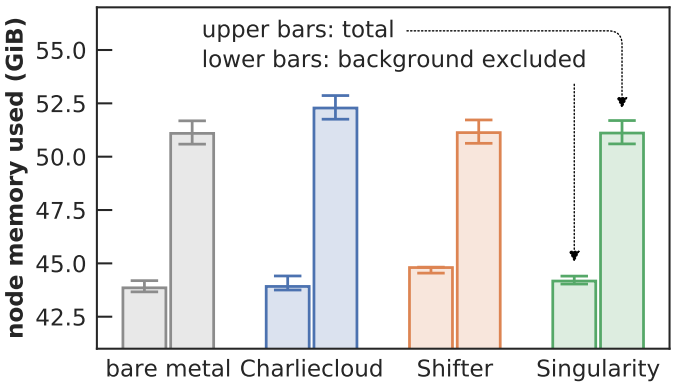

Fig. 5. STREAM memory usage sampled from /proc/meminfo; lower is better. Bar height is median of all samples; error bars are minimum and maximum. Taller bars are total node memory usage, while shorter bars have background usage (before STREAM start) removed. Usage is broadly similar across the four environments, but there are a few differences described in the text.

that the bare metal executable again had additional shared libraries linked to it. We speculate that this is the cause of the slightly slower performance on bare metal.

All conditions showed very linear weak scaling for HPCG, i.e., there is no performance degradation as the problem size and number of nodes increases, so we have omitted this discussion.

In summary, while these tests showed considerably larger performance differences between the environments than the previous two, they are still relatively small, and we believe we have good explanations unrelated to containerization. We believe these results support the hypothesis of negligible performance overhead of HPC container runtimes.

D. Memory usage

Recall that we measured memory usage in three different ways. Figure 5 addresses total node memory and container implementation overhead, showing memory usage for STREAM sampled from /proc/meminfo, specifically MemTotal minus MemFree; lower is better (less memory used). We removed the small number of samples less than 40 GiB to exclude startup and teardown effects.

The taller bars show median total node memory used. At median, Charliecloud adds 1200 MiB over bare metal, while Shifter and Singularity add an insignificant 16 MiB and 37 MiB respectively. Charliecloud's memory cost is most likely due to its complete 1.2 GiB image residing in memory in a tmpfs. This cost could be greatly reduced or eliminated by Charliecloud's newer image workflow, which mounts SquashFS image files rather than unpacking into a tmpfs.

The shorter bars show node memory used after background is removed. Under this metric, Charliecloud adds only 66 MiB at median, Shifter 970 MiB, and Singularity 330 MiB. We believe this is additional memory used by the container implementation itself, and implementors should verify our results and explore whether there are any efficiency gains available.

Figure 6 shows histograms that describe memory used by HPCG processes at four node counts, sampled from the writeable/private field of pmap; leftward is better (less memory used). This measures memory used by the application itself. In this test, the four environments overlap closely, showing almost no difference. This result suggests that containerization does not alter memory behavior of applications.

Overall, we find that memory overhead for the container technologies is low, at most 1–2% of node memory for our experiments, and no implementation is clearly more memory-efficient than the others.

V. Implications, limitations, and future work

This work evaluates performance impact of the three key HPC container technologies on multiple dimensions using industry-standard benchmarks at HPC scale, a combination of rigor not previously available in the literature.

These technologies use diverse approaches to containerization. In addition to their diverse feature sets, Charliecloud is implemented mostly in C, shell, and Python; Shifter in C, Python, and C++; and Singularity in Go. Charliecloud as tested unpacks into a tmpfs, Shifter mounts a SquashFS image file, and Singularity mounts an ext4 image file. Charliecloud uses user plus mount namespaces, Shifter mount only, and Singularity mount plus UTS (host and domain name). Shifter and Singularity are setuid, while Charliecloud is not.

Despite these implementation differences, we found that performance impact of containers is minimal to nonexistent, though there is a modest memory overhead. More specifically, our results suggest:

1) Performance impact is not only low, but it may actually be zero. That is, *we hypothesize that the performance impact of containerization itself is nil*. However, differing software environments do impact performance; we hypothesize that the performance differences previously seen for containerization, both higher and lower, anecdotal and experimental, are the result of differing environments. For example, in the present work, we saw noticeable performance degradation of 1.8% when SysBench was linked with apparently-irrelevant libaio. Containerization makes it easy to produce differing environments, intentionally or not, and even subtle differences matter.
2) Memory impact is minimal (hundreds of MiB), but certain workflows, such as unpacking images into memory, scale with the size of the image and should be used with care. Vendors should understand their memory impact and be prepared to justify it.
3) Container implementations have no meaningful differences in terms of performance and minimal differences in terms of memory impact, and the latter is likely to change as development proceeds. Performance is not a reason to choose one implementation over another.

All experiments are imperfect, and our results should be understood in the context of both their advantages and limitations. The latter include:

- We did not evaluate startup or teardown overhead. We believe, however, that this would not meaningfully alter

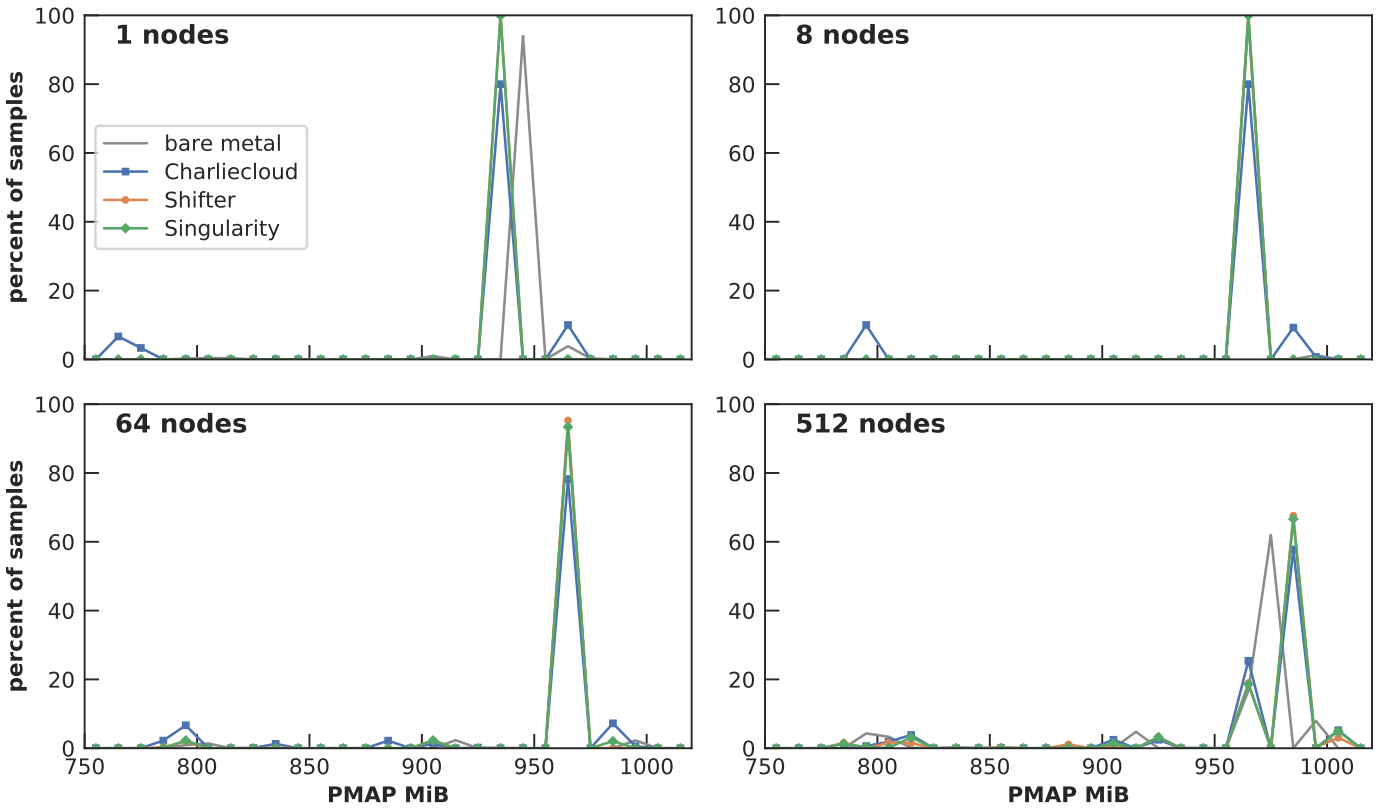

Fig. 6. HPCG memory usage sampled from pmap; leftward is better. This is a histogram of HPCG ranks and measures memory used by the application itself. The four environments showed very similar usage.

our conclusions. For example, Charliecloud in the tested configuration likely has the largest startup time because the image needed to be unpacked from its tarball, but this took less than 11 seconds on average, even at 512 nodes.

- We did not evaluate the common performance trick of bind-mounting host libraries into the container image [5]. This is necessary when host libraries are required to access proprietary hardware (e.g., Cray networks) or closely match the host kernel (nVidia GPUs). Some anecdotal reports also report that host-tuning of MPI libraries is important. Our focus for the present work was understanding the impact of containerization and container implementations, not library differences.

We believe there is more work ahead, but that container performance is closed to a solved question. Remaining problems include (a) testing a more comprehensive suite of benchmarks, including filesystems, networking, MPI, real applications, container image formats (e.g. tarballs, SquashFS and other filesystem images, layered), and more runtimes (e.g., Podman and enroot); (b) an enumeration of performance gotchas, such as huge pages and startup/teardown; and (c) evaluation at full scale on the largest HPC systems.

Our results build upon prior work to reassure users and sysadmins that the flexibility gained by using containers does not come at the cost of performance. Containerization is an important tool that should be used when it serves the science.

Acknowledgements

Anonymous reviewers and Andrew Younge provided important feedback that materially improved this paper.

References

[1] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in HPC," in *Proc. SC*, 2017.
[2] D. M. Jacobsen and R. S. Canon, "Contain this: Unleashing Docker for HPC," in *Proc. CUG*, 2015.
[3] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, 2017.
[4] M. G. Xavier *et al.*, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proc. PDP*, 2013.
[5] A. J. Younge *et al.*, "A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds," in *Proc. CloudCom*, 2017.
[6] W. Felter *et al.*, "An updated performance comparison of virtual machines and Linux containers," in *Proc. ISPASS*, 2015.
[7] M. Chae, H. Lee, and K. Lee, "A performance comparison of Linux containers and virtual machines using Docker and KVM," *Cluster Computing*, 2017.
[8] A. Acharya *et al.*, "A performance benchmarking analysis of hypervisors containers and unikernels on ARMv8 and x86 CPUs," in *Proc. EuCNC*, 2018.
[9] D. Brayford *et al.*, "Deploying AI frameworks on secure HPC systems with containers," in *Proc. HPEC*, 2019.
[10] Ákos Kovács, "Comparison of different Linux containers," in *Proc. TSP*, 2017.
[11] E. Le and D. Paz, "Performance analysis of applications using Singularity container on SDSC Comet," in *Proc. PEARC*, 2017.
[12] Y. Wang, R. T. Evans, and L. Huang, "Performant container support for HPC applications," in *Proc. PEARC*, 2019.